Introduction to Object-Oriented Programming

Conditional Execution

Christopher Simpkins
chris.simpkins@gatech.edu

Structured Programming

In reasoning about the flow of control to and from a statement, consider control flow issues:

- Multiple vs. single entry ("How did we get here?")
- Multiple vs. single exit ("Where do we go from here?")
- `goto` considered harmful (`goto` makes it hard to answer questions above)

Structured programming: block structure, single entry, single exit, no `goto`. All algorithms expressed by:

- Sequence - one statement after another
- Selection - conditional execution (not conditional jumping)
- Iteration - loops

Today we'll learn Java's support for conditional execution

Boolean Values

There are 10 kinds of people:

- Those who know binary,
- and those who don't.

Boolean Values

There are 10 kinds of people:

- Those who know binary,
- and those who don't.

Boolean Values

There are 10 kinds of people:

- Those who know binary,
- and those who don't.

Boolean Values

In Java, boolean values have the `boolean` type. Four kinds of boolean expressions:

- `boolean` literals: `true` and `false`
- `boolean` variables
- expressions formed by combining non-`boolean` expressions with comparison operators
- expressions formed by combining `boolean` expressions with logical operators

Boolean Expressions Formed From Comparisons

Simple boolean expressions formed with comparison operators:

- Equal to: ==, like $=$ in math
  - Remember, = is assignment operator, == is comparison operator!
- Not equal to: !=, like $\neq$ in math
- Greater than: >, like $>$ in math
- Greater than or equal to: >=, like $\geq$ in math
- ...

Examples:

```
1 == 1 // true
1 != 1 // false
1 >= 1 // true
1 > 1  // false
```

Boolean Expressions Formed From Logical Operators

Simple boolean expressions can be combined to form larger expressions using:

- And: `&&`, like $\wedge$ in math
- Or: `||`, like $\vee$ in math

Examples:

```
(1 == 1) &&  (1 != 1) // false
(1 == 1) ||  (1 != 1) // true
```

Also, unary negation operator `!`:

```
!true      // false
!(1 == 2) // true
```

The `if-else` Statement

Conditional execution:

```
if (booleanExpression)
    // a single statement executed when booleanExpression is true
else
    // a single statement executed when booleanExpression is false
```

- *booleanExpression* must be enclosed in parentheses
- `else` not required

Example:

```
if ((num % 2) == 0)
    System.out.printf("I like %d.%n", num);
else
    System.out.printf("I'm ambivalent about %d.%n", num);
```

Ternary If-Else Expression

The ordinary `if-else` control structure is a statement, leading to conditional assignment code like this:

```
String dinner = null;
if (temp > 60) {
    dinner = "grilled";
} else {
    dinner = "baked";
}
```

The ternary operator combines the above into one expression (recall that expressions have values):

```
String dinner = (temp > 60) ? "grilled" : "baked"
```

Blocks

Java is block-structured. You can enclose any number of statements in curly braces ({ ... }) to create a block. Blocks are like single statements (not expressions - they don't have values).

```
if ((num % 2) == 0) {
    System.out.printf("%d is even.%n", num);
    System.out.println("I like even numbers.");
} else {
    System.out.printf("%d is odd.%n", num);
    System.out.println("I'm ambivalent about odd numbers.");
}
```

The Java conventions recommend using braces always, even for single statements. A very common error is adding statements to an if-branch and forgetting to add braces.

Multi-way `if-else` Statements

This is hard to follow:

```
if (color.toUpperCase().equals("RED")) {
    System.out.println("Redrum!");
} else {
    if (color.toLowerCase().equals("yellow")) {
        System.out.println("Submarine");
    } else {
        System.out.println("A Lack of Color");
    }
}
```

This multi-way `if-else` is equivalent, and clearer:

```
if (color.toUpperCase().equals("RED")) {
    System.out.println("Redrum!");
} else if (color.toLowerCase().equals("yellow")) {
    System.out.println("Submarine");
} else {
    System.out.println("A Lack of Color");
}
```

Short-Circuit Evaluation

Here's a common idiom for testing an operand before using it:

```
if ((kids !=0) && ((pieces / kids) >= 2))
    System.out.println("Each kid may have two pieces.");
```

In this example Java uses short-circuit evaluation. If

```
kids !=0
```

evaluates to `false`, then the second sub-expression is not evaluated, thus avoiding a divide-by-zero error.

Note: You can force a complete evaluation by using `&` or `|`, for example if you have side effects you want to ensure happen in the second expression. We mention this fact for completeness but implore you not to write such code.
See Conditionals.java for examples.

The `switch` Statement

Java provides `switch` statement for multi-way branching.

```
switch (expr) {
case 1:
    // executed only when case 1 holds
    break;
case 2:
    // executed only when case 2 holds
case 3:
    // executed whenever case 2 or 3 hold
    break;
default:
    // executed only when other cases don't hold
}
```

- Execution jumps to the first matching case and continues until a `break`, `default`, or `switch` statement's closing curly brace is reached
- Type of `expr` can be `char`, `int`, `short`, `byte`, or `String`
- In example above, what is type of `expr`?

Avoid the `switch` Statement

The `switch` statement is error-prone.

- `switch` considered harmful. 97% of fall-throughs unwanted[1]
- Anachronism from "structured assembly language", a.k.a. C (a `switch` is just a jump table)

You can do without the switch statement. See

- CharCountSwitch.java for a `switch` example,
- CharCountIf.java for the same program using an `if` statement in place of the `switch` statement, and
- CharCount.java for the same program using standard library utility methods.

[1] Peter van der Linden, *Deep C Secrets*

Closing Thoughts

- Conditional execution straightforward, but watch out for side-effects in boolean assignments.
- Parenthesize your expressions to make them clear to the reader and to Java.
- Next we'll learn loops, and we'll have all the tools we need to implement any algorithm.[2]

[2] Actually we already have all the tools we need to implement any algorithm in a functional style, but we need loops for imperative algorithms.